An Efficient OpenMP Loop Scheduler for Irregular Applications on Large-Scale NUMA Machines

Marie Durand[1], François Broquedis[2], Thierry Gautier[1], and Bruno Raffin[1]

[1]INRIA, [2]Grenoble Institute of Technology
MOAIS Team, Computer Science Laboratory of Grenoble, France
marie.durand@inria.fr, françois.broquedis@imag.fr,
thierry.gautier@inrialpes.fr, bruno.raffin@inria.fr

Abstract. Nowadays shared memory HPC platforms expose a large number of cores organized in a hierarchical way. Parallel application programmers struggle to express more and more fine-grain parallelism and to ensure locality on such NUMA platforms. Independent loops stand as a natural source of parallelism. Parallel environments like OpenMP provide ways of parallelizing them efficiently, but the achieved performance is closely related to the choice of parameters like the granularity of work or the loop scheduler. Considering that both can depend on the target computer, the input data and the loop workload, the application programmer most of the time fails at designing both portable and efficient implementations. We propose in this paper a new OpenMP loop scheduler, called *adaptive*, that dynamically adapts the granularity of work considering the underlying system state. Our scheduler is able to perform dynamic load balancing while taking memory affinity into account on NUMA architectures. Results show that *adaptive* outperforms state-of-the-art OpenMP loop schedulers on memory-bound irregular applications, while obtaining performance comparable to *static* on parallel loops with a regular workload.
Keywords: *OpenMP, NUMA, loop scheduling, runtime systems.*

1 Introduction

Large-scale multicore platforms are commonly used by the HPC community. They expose a constantly-growing number of cores organized in a hierarchical way, leading to large-scale NUMA machines. To exploit them at their full potential, the application programmer needs to express massive fine-grain parallelism while taking memory affinity into account. Applications exposing irregular workloads are really difficult to execute efficiently on such platforms, as they require to deal with both load balancing and memory locality.

OpenMP [17], the *de-facto* standard for shared memory parallel programming, provides the programmer with high-level constructs to ease the parallelization of serial applications. The parallel loop certainly remains the most widely used OpenMP construct, allowing to easily parallelize loops with independant iterations. The end-user can control the way loop iterations are assigned to OpenMP threads by invoking OpenMP loop schedulers. Choosing the best scheduler for a specific parallel loop can be a difficult task to perform in a portable way [1]. The application programmer is also responsible for defining the granularity of work within the loop, using the `chunk_size` clause.

While being designed to tackle loops with irregular workloads, the OpenMP `dynamic` scheduler suffers from two main issues on large-scale NUMA machines.

First, defining a chunk size from the application level that achieves both high and portable performance has never been so difficult. Indeed, parallel loops with big chunks may suffer from load imbalance, while the ones with smaller chunks are more sensitive to runtime-related overheads which are getting more and more noticeable as the number of cores per NUMA node increases.

Second, traditional techniques to increase the performance of memory bound applications, like guiding the data allocation on the different NUMA nodes of the platform, are useless using dynamic scheduling, as the assignment of loop iterations to OpenMP threads is non-deterministic.

We introduce in this paper a new loop scheduler, called *adaptive*, that outperforms state-of-the-art loop schedulers when executing memory bound irregular applications. In particular, our loop scheduler:

1. dynamically adapts the granularity of work within parallel loops according to the machine resources utilization,
2. relies on data placement information to guide load balancing on NUMA platforms.

The remainder of the paper is organized as follows. We first describe the related work on loop scheduling over NUMA architectures in section 2. Then we introduce the *adaptive* loop scheduler and the way we implemented it inside the LIBGOMP library in sections 3 and 4. Eventually, we report the performance we obtained on several benchmarks and applications in section 5 before concluding.

2 Related Work

Many research projects have been carried out to improve execution of OpenMP applications on NUMA machines.

The HPCTools group at the University of Houston has been working in this area for a long time, proposing compile-time techniques that can help improving memory affinity on hierarchical architectures like distributed shared memory platforms [13]. Huang et al. [10] proposed OpenMP extensions to deal with memory affinity on NUMA machines, like ways of explicitly aligning tasks and data inside logical partitions of the architecture called *locations*. While the proposed extension is interesting to deal with regular memory-bound applications, it does not tackle the problems induced by irregular workloads.

Olivier et al. [16, 15] introduced node-level queues of OpenMP tasks, called *locality domains*, to ensure tasks and data locality on NUMA systems. The runtime system does not maintain affinity information between tasks and data during execution. Data placement is implicitly obtained considering that the tasks access memory pages that were allocated using the *first-touch* allocation policy. The authors thus ensure locality by always scheduling a task on the same locality domain, preventing application programmers to experiment with other memory bindings.

The INRIA Runtime group at the University of Bordeaux proposed the Forest-GOMP runtime system [2] that comes with an API to express affinities between

OpenMP parallel regions and dynamically allocated data. ForestGOMP implements load balancing of nested OpenMP parallel regions by moving branches of the corresponding tree of user-level threads on a hierarchical way. Memory affinity information is gathered at runtime and can be taken into account when performing load balancing. Our work extends this approach to deal with parallel loops while ensuring load balancing in a different way.

Mahéo et al. [12] used similar techniques to speed up hybrid MPI/OpenMP synchronizations on hierarchical architectures, including NUMA machines. Both our work and theirs build upon the same concepts and can be stated as complementary.

Subramanian and Eager [18] introduce an affinity loop scheduler for unbalanced workload. They study iterative applications involving series of parallel loops in which *"the execution times of any particular iteration do not vary widely from one execution of the loop to the next"*[18]. Their proposition is based upon a two-phase algorithm: first, the iterations are equally divided between the processors; then the scheduler dynamically adapts the workload by making idle processors steal a constant fraction $(1/P)$ of the remaining iterations from occupied ones. In [21], Yong *et al.* extends the work of Subramanian and Eager by providing new ways of adapting the workload considering an history of previous executions of a particular parallel loop.

Taking advantage of the temporal coherency of the simulation is a very interesting idea but cannot be used in all situations. For instance, it would not be effective on applications involving several parallel loops with varying workloads, like the PMA application we used to evaluate our *adaptive* loop scheduler. The first phase of our scheduling strategy is similar to the one introduced by Subramanian *et al.*, as *adaptive* equally pre-distributes the loop iterations over the processors, which is a compromise between balancing the workload of the loop and preserving the affinity across several executions of the same loop. However, the second phase of our algorithm is different from the one implemented by Subramanian's adaptive loop scheduler. Indeed, *adaptive* relies on a work-stealing algorithm [8] in which idle processors steal half of the remaining iterations from a randomly-selected victim. With such an approach, our *adaptive* loop scheduler does not require to maintain a global view of the workloads associated to each processor, unlike proposition [18].

3 Introducing the Adaptive Loop Scheduler

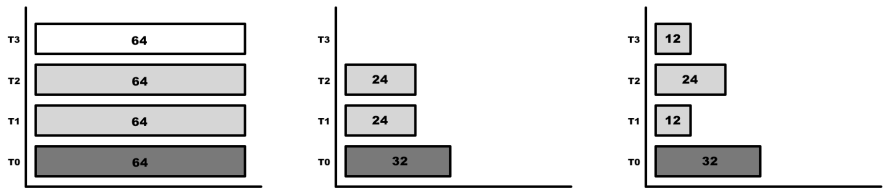
The OpenMP programmer can rely on two main loop schedulers to specify the way loops iterations should be assigned to threads. The first one, called *static*, statically assigns fixed portions of work in a round-robin fashion. This scheduler behaves well on loops with a regular workload and is often used in the context of NUMA architectures, along with the *first-touch* allocation policy, to maximize memory locality. The second one, called *dynamic*, makes OpenMP threads steal fixed portions of work from a centralized queue. This scheduler behaves better than *static* on loops with an irregular workload. However, *dynamic* is seldom used on NUMA architectures because of its non-deterministic behavior, preventing the programmer from controlling memory locality.

The loop scheduler we propose goes beyond those two, providing ways of balancing the load of irregular loops while respecting memory locality. This section first introduces the main scheduling algorithm provided by our *adaptive* loop scheduler before presenting the way we extended it to deal with memory locality on NUMA machines.

3.1 Designing an OpenMP Loop Scheduler with Adaptive Granularity

Dealing with irregular parallel applications requires efficient runtime-level functionalities to perform dynamic load balancing with the lowest overhead possible. OpenMP application programmers can rely on the *dynamic* loop scheduler to execute loops with irregular workload, as long as they manage to specify a chunk size that achieves good performance. Indeed, a too small chunk size will increase the time spent inside the runtime system, while a too coarse chunk size will limit the potential parallelism and the ability to balance the work load.

We adopted a different approach implementing our *adaptive* scheduler. We relieve the application programmer from statically deciding the granularity of work which can lead to non-portable solutions. Instead, we consider work-stealing as an oblivious technique to dynamically balance the load on the threads of the corresponding OpenMP parallel region.



(a) Initial distribution of the iteration space.

(b) T3 has completed its range and starts stealing.

(c) T3 stole half of the remaining iterations of T1.

Fig. 1: Illustration of adaptive loop scheduling on a 256-iterations loop with irregular workload executed on 4 threads. Darker color means higher workload.

We broke the vision of centralized work used by the state-of-the-art OpenMP loop scheduler to introduce a per-thread data structure describing the range of iterations assigned to each thread. Figure 1 illustrates the behavior of our *adaptive* scheduler on a synthetic example. Considering a loop of **imax** iterations executed by **nthr** threads, the scheduler first assigns **imax / nthr** iterations to each thread, like presented on figure 1a. This initial behavior allows our scheduler to achieve performance that is comparable to *static* on loops with regular workload. Even if every thread has the same number of iterations to execute here, the associated workload is different: the 64 iterations of thread **T3** have shorter execution times than the ones assigned to **T2** for example. This leads to load imbalance: at some point of the execution of the loop, thread **T3** will be starving like showed on figure 1b. **T3** will then trigger *adaptive*'s work-stealing algorithm which steals half of the remaining iterations of a loaded thread.

```
iter_adaptive_next:
1  if (try_local_work (&begin, &end) == true)
2    return (begin, end);
3
4  /* We've completed our previously-assigned range,
5  let's try to steal some new work! */
6  while (!loop_is_finished()) {
7    victim = pick_random_victim ();
8    if (steal_from_victim (victim, &begin, &end) == true)
9      return (begin, end);
10 }

steal_from_victim:
11 if ((victim->end-victim->begin)>0){
12  lock (victim);
13  chunk_size
     =(victim->end - victim->begin)/2;
14  end = victim->end - chunk_size;
15  victim->end = end;
16  if (end <= victim->begin) {
      /* rollback and abort */
17    victim->end = end + chunk_size;
18    unlock (victim)
19    return false;
20 }
21  begin = end;
22  end = begin+chunk_size;
23  unlock (victim);
24  return true;
25 }
26 return false;

try_local_work:
27 begin = own->begin +1;
28 own->begin = begin;
29 if (begin < own->end) {
30   end = begin;
31   begin = end - 1;
32   return true;
33 }
34 /*conflict detected: rollback and lock*/
35 own->begin = begin-1;
36 lock (self)
37 begin = own->begin;
38 if (begin < own->end)
39   end = own->begin = begin + 1;
39 unlock (self);
40 if (begin < own->begin) return true;
41 return false;
```

Fig. 2: Pseudo-code of the adaptive loop scheduler. The implementation extends the THE protocol by stealing more than one item at each steal operation.

The algorithm[1] called to select the next chunk of iterations to execute is summarized on figure 2. Most importantly, our approach deals with dynamic per-thread chunk sizes, as shown on line 13 of this pseudo-code. The amount of work a thread will steal depends on the amount of work its victim still has to execute, unlike the *dynamic* scheduler in which the *chunk_size* is statically defined and cannot change during execution.

Unlike [18, 21] in which a constant fraction of the work $(1/P)$ is removed from the most loaded processor, the random selection of the work-stealing algorithm does not suffer from maintaining the global state of the workloads associated to processors. Moreover, it is possible to derive theoretical performance guarantee of scheduling parallel loop with work stealing [20]. Frigo *et al.* [8] introduced two main metrics to model the performance of work-stealing-based algorithms: the *work* W, *i.e.* the time to sequentially execute the loop, and the *depth* D, also called the critical path, *i.e.* the execution time on an infinite number of processors.

Considering these metrics, the average completion time of the parallel loop is $O(W/P + D)$. If the work is $W = \sum_{i=0}^{n-1} w_i$, where w_i is the work associated to the i-th iteration, then $D = O(\log n + \max\{w_i\})$.

3.2 Extending the Adaptive Scheduler to Deal with Locality

Ensuring memory locality is crucial to achieve good performance on NUMA architectures. We extended the *adaptive* scheduler in order to benefit from shared memory

[1] If the memory consistency is not sequential, memory barriers have to be inserted between lines 15-16 and 28-29.

```
/* get the number of locality domains of a parallel region */
#pragma omp parallel
#pragma omp master
      printf("Number of locality domains = %i\n",
          omp_get_num_locality_domains());

/* sample of the modified STREAM benchmark with the adaptive
 scheduler and bloc memory distribution on locality domains */
#pragma omp parallel
#pragma omp master
    a = (double*)omp_locality_domain_allocate_bloc1d(
                    sizeof(double)*STREAM_ARRAY_SIZE+OFFSET);
```

Fig. 3: Code sample using our extended OpenMP runtime APIs.

banks of NUMA multicore machines. This is done at two levels. First, we make the cores attached to the same memory bank work on contiguous iterations. This step is useful to abstract the OS identification of cores that may not be contiguous on a NUMA node. *adaptive* makes idle threads steal work from cores that belong to the same NUMA node. Thanks to this strategy, a successful steal will hopefully reduces the number of remote memory transfers. Moreover, this local steal strategy may not be enough to balance the workload among all the cores. That is why, if the number of unsuccessful steal operations reaches a threshold, the idle thread emits a steal request to a victim randomly selected over the whole machine.

The second feature is to provide ways of distributing the application data over the NUMA nodes. Our extension of the libGOMP library comes with APIs to distribute data as proposed in MAMI [4], implementing bloc and bloc-cyclic data distributions as runtime extensions. For now, we only support data distribution within parallel regions where the participating threads are bound to cores, for example using the GOMP_CPU_AFFINITY environment variable. As in [10, 16, 15], we refer to NUMA nodes as *locality domains*.

omp_get_num_locality_domains() : returns the number of locality domains holding threads from the current OpenMP parallel region.

omp_get_locality_domain_num() : returns the locality domain of the running thread.

omp_locality_domain_allocate_bloc1dcyclic (size, blocsize) : allocates an array of `size` bytes in a `blocsize`-cyclic distribution over the locality domain of the parallel region.

omp_locality_domain_allocate_bloc1d (size) : allocates an array of `size` bytes using a bloc distribution over the locality domain of the parallel region.

These routines performs allocation following the OS constraints: sizes are rounded up to a multiple of page size. If the OS does not support NUMA allocation routines, the implementation triggers calls to the libc's malloc function.

Figure 3 illustrates the use of the runtime APIs we propose. In order to allow reuse of memory mapping across parallel regions, we ensure that the $i-th$ thread of a parallel region will be bound to the same core across parallel regions if and only if the following parallel regions have the same size and are nested in the same parallel region or are at the top level.

3.3 Discussion

Defining the best granularity of work is certainly one of the most difficult challenge a parallel application programmer has to face to exploit HPC platforms at their full potential. For example, the best *chunk_size* for a specific OpenMP loop may depend on the target architecture and the input data of the parallel application. In other words, application programmers have to consider the underlying system state to specify the granularity that will achieve the best performance. This is an old problem for the OpenMP community: defining the *right* number of threads, the *right* level of nested parallel regions and the *right* chunk size for parallel loops are a few examples of the many crucial steps to achieve good performance and scalability.

The addition of tasks to the OpenMP standard provides new ways of expressing parallelism with a finer granularity. One can consider tasks as another way of dealing with irregular workload, as tasks can move from one OpenMP thread to another to perform load balancing. However, tasks will not solve the problem of granularity, as defining the *right* number of tasks can be challenging, as studied in our previous work [3].

Our proposal introduces a runtime-level approach to deal with granularity and has been implemented inside a loop scheduler. The same approach could be applied to task parallelism as well, considering ways of *splitting* OpenMP tasks when necessary. Our group has carried out research in this context [19] that could be extended to OpenMP. The application programmer could provide functions to split a running task into smaller ones, similarly to the way our adaptive loop scheduler splits ranges of iterations. This idea was applied to more general iterative algorithms where dependencies may exist between iterations [20].

4 Implementation Details on Extending libGOMP with Adaptive Loop Scheduling

We implemented our *adaptive* loop scheduler inside the original LIBGOMP library that comes with GCC 4.6.2. Our loop scheduler can be experimented with parallel loops stated as `schedule(runtime)` by setting the `OMP_SCHEDULE` environment variable to `"adaptive,chunk_size"` before running the application. This allows us to experiment with our proof-of-concept implementation without modifying the compiler.

The implementation (figure 2) of the stealing mechanism used in the adaptive loop scheduler is greatly inspired from Cilk's THE algorithm [8] designed to limit the perturbation of the serial execution from stealing-related overheads. Unlike other OpenMP loop schedulers, *adaptive* uses a per-thread data structure describing the range of iterations assigned to the considered thread. This structure basically contains the boundaries of this range (*[begin, end)*) and an atomic-based lock used to synchronize the stealing thread and its victim. Each thread pops *chunk_size* iterations to execute out of its own range (*begin += chunk_size*), until there are no more iterations left (*begin == end*). Stealing a range of iterations from a busy thread is simply a matter of shrinking the *end* bound of the victim's data structure down to *end - N*, *N* being the number of iterations we want to steal. The THE algorithm uses an optimistic approach to minimize the need for a thread to lock its own data structure on a pop operation. This can be done

chunk size	1	2	4	8	16	32	64	128
static	30.44	28.20	25.97	25.03	24.40	24.50	24.18	24.50
dynamic	1328.43	594.30	232.61	75.43	36.29	35.20	34.02	33.21
guided	77.86	69.49	55.55	45.47	42.90	43.16	58.66	30.54
adaptive	55.92	50.74	48.26	47.72	47.69	47.97	48.90	49.44
adaptive (no steal)	30.29	27.48	25.67	24.48	25.21	24.48	24.16	23.23

Table 1: Overhead measured by EPCC benchmark (in μs) of the *adaptive* loop scheduler versus *static*, *dynamic* and *guided* on the AMD48 platform.

by detecting conflicting accesses to the same data structure, by comparing the value of *end* before and after the pop operation. If the value has changed, someone accessed the data structure during the pop: the thread will then undo this last pop and acquire the lock before trying again. Such implementation greatly minimizes the overhead added to threads performing local work (Cilk's work first principle [8]).

The memory binding routines rely on *libNUMA* and the `mbind` system call. The current prototype was developed on Linux. libGOMP maintains a pool of threads (`gomp_thread` and `gomp_thread_pool`) attached to each parallel region. We extended the data structures in order to maintain simple per-thread NUMA-related information, like the core id, the NUMA node id and a list of threads per NUMA node that can be used by the *adaptive* scheduler to select a victim. Based on this information, the scheduler initializes of per-loop data structure to balance the iterations over the NUMA nodes taking the number of cores per NUMA node into account.

5 Performance Evaluation

We conducted our experiments on two different ccNUMA configurations.

The first one holds 8 AMD Magny Cours processors for a total of 48 cores. Each core has access to 64 KB of L1 cache, 512 KB of L2 cache. Both L1 and L2 caches are private, while L3 cache is shared between the 6 cores of a processor. This configuration provides a total of 256 GB (32 GB per NUMA node) of main memory. We will refer to this configuration as **AMD48** in the following of the paper.

The second configuration holds 12 groups containing two Intel Sandy Bridge processors each for a total of 192 cores. 32 GB of main memory is attached to each socket, for a total of 768 GB. Inter-groups communications use the SGI NUMAlink technology, while standard Intel QuickPath interconnect provides inner-group communications. We will refer to this configuration as **Intel192** in the following of the paper.

All experiments were performed with the libGOMP library distributed with GCC 4.6.2.

5.1 EPCC: Overhead of the Adaptive Loop Scheduler

The EPCC benchmark [5] reports runtime-related overheads when performing OpenMP loop scheduling with respect to the corresponding serial execution. The measured overheads of the four loop schedulers are reported in table 1 for different chunk sizes. A

larger chunk size implies less calls to the runtime and thus a smaller overhead. Reported measures reported represent the average performance over 10 runs. The libGOMP implementation of both the *dynamic* and the *guided* schedulers suffer from a high overhead for the three smallest chunk size values tested in this experiment.

The *adaptive* scheduler adds an extra overhead to $25\mu s$ with respect to the *static* scheduler. By disabling the steal operations from the *adaptive* scheduling algorithm, we are able to provide finer estimation of the overheads. The performance of this modified scheduler, named *adaptive (no steal)* in the table, reports no overhead induced by the initial work distribution over the *static* scheduler. We can thus infer that an extra $25\mu s$ includes the overheads of the work-stealing operations and the detection of the termination.

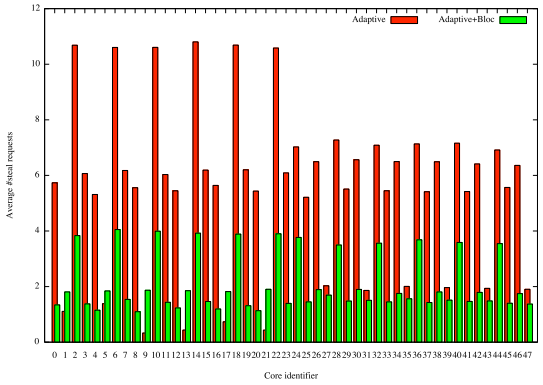
Obviously, our code is not as optimized as the other scheduler implementations from libGOMP. We will add some optimizations (memory alignment of data structures, lazy initialization) to reduce overheads, and we believe that those may improve the performance of all libGOMP schedulers as well.

5.2 STREAM: Impact of the Memory Hierarchy

The STREAM benchmark [14] measures the maximal achievable bandwidth over four memory bound kernels (`copy`, `scale`, `add` and `triad`). We evaluated the behavior of the *static* and the *adaptive* schedulers with two memory allocation strategies: a *first-touch* strategy and an explicit bloc distribution of the arrays over the 8 NUMA nodes of the AMD48 platform using the API presented in section 3.2 . The memory per array is 150.0 MB and the number of iterations is set to 500. Measures are reported in table 4a.

	static		*adaptive*	
	first-touch	bloc	first-touch	bloc
Copy	6.9	6.9	6.4	6.8
Scale	6.7	6.8	6.2	6.7
Add	7.2	7.4	6.8	7.3
Triad	6.9	7.5	6.6	7.4

(a) Achieved bandwidth (GB/s) reported by the STREAM benchmark for the *static* and *adaptive* loop schedulers.



(b) Number of steal operations with the *adaptive* scheduler with and without bloc memory allocation.

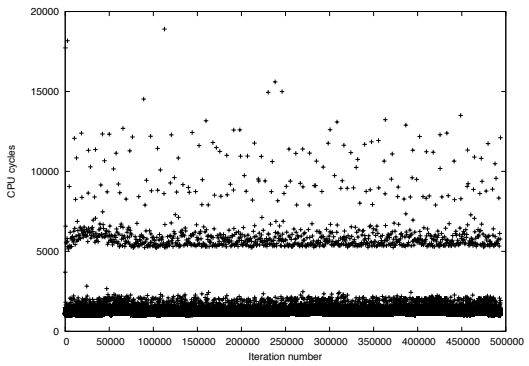
Fig. 4: Performance evaluation of the STREAM benchmark on the AMD48 platform.

On the Triad kernel, the *bloc* allocation strategy increases the performance of at most 7% with the *static* scheduler and of at most 12% with the *adaptive* scheduler. The

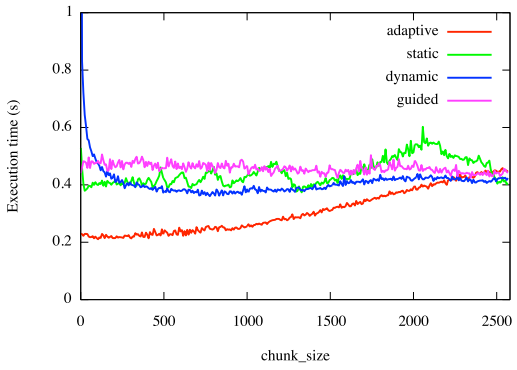
two schedulers reach comparable performances on this highly regular benchmark, with a slight advantage for *static* over *adaptive*.

This difference comes from the steal operations performed by the *adaptive* scheduler. Figure 4b shows the average number over 500 iterations of successful steal requests per core. We can see that the *adaptive* scheduler with the NUMA-aware extension taking the memory distribution into account helps reducing the number of steal operations.

On this memory-intensive benchmark, *adaptive* is able to reach performances that are similar to *static* as the load balancing strategy first conside the cores from the same NUMA node to perform work-stealing, thus favoring memory locality on such architectures.



(a) Execution time, in CPU cycles, of each iteration of the kmeans kernel main loop on a single core of the Intel192 platform.



(b) Average performance of different OpenMP loop schedulers for varying values of chunk_size on the Intel192 platform.

Fig. 5: Performance evaluation of the K-Means benchmark on the Intel192 platform.

Time in $m.s$	static	dynamic chunk=1	dynamic chunk=3000	guided	adaptive
numactl	17.8	57.4	12.1	15.2	11.6
bloc distribution	16.3	57.2	14.2	14.9	6.95

Table 2: Comparison of the four loop schedulers on PMA on the AMD48 platform.

5.3 K-Means: Benefits of Adaptive Granularity for OpenMP Loops

We evaluated the *adaptive* loop scheduler with the OpenMP version of the K-Means kernel coming from the Rodinia benchmark suite [6]. This kernel implements a clustering algorithm commonly used by data-mining applications. Its parallel implementation involves an OpenMP parallel loop with an irregular workload.

Figure 5a reports the execution times of each one of the 494020 iterations of this loop executed on a single core of the Intel192 platform. We can distinguish at least two main classes of iterations on this figure with different execution times, but we can only consider this as a rough source of information, as the execution times of the same iterations may vary when executing them in parallel, depending for example on the capacity of threads to efficiently communicate through shared cache memory.

The results obtained running this kernel with the *adaptive* scheduler confirms K-Means can benefit from dynamic load balancing. Figure 5b shows a performance comparison between the *adaptive*, *static*, *dynamic* and *guided* schedulers on the K-Means kernel. We experimented with different values for the *chunk_size* clause of the for loop. These tests were executed on the 192 cores of the Intel192 platform. We tested every value of *chunk_size* from 1 to 2574, corresponding to **imax / nthr** here. The best performance is achieved by our scheduler. *adaptive* performs especially well for executions with small chunk sizes, as they offer more options to perform load balancing. We can also note that, even if the workload of this kernel is irregular, the best performance of the *dynamic* scheduler can almost be achieved by *static* for a tuned value of *chunk_size*.

5.4 PMA: Dealing with Both Load Balancing and Locality

We applied our *adaptive* loop scheduler to a practical situation in physical simulations considering elements of a 3D space that evolve with respect to physical laws. Maintaining these elements ordered is a key factor to improve memory efficiency as elements are likely to interact with their neighbors [11].

The Packed Memory Array (PMA) [7] data structure has been proposed to help maintaining its elements ordered in an efficient way. This sparse data structure was designed to reduce the amount of memory movement induced by reordering operations.

We focus on the loop that handles both the detection of the moving elements and their copy in a dense array. In real applications, the workload gets irregular since some parts of the physical space go through a lot of changes while others report only a few changes. We extracted actual change distribution from a memory-intensive fluid simulation [9] ran with $2\,900\,000$ elements.

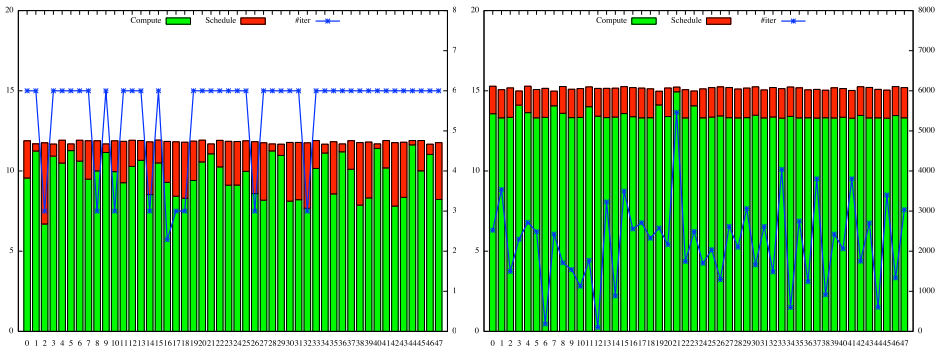
In this code, the data structure is initialized from reading sequentially a file. Without major rewrite of the initialization phase, it is not possible control the

affinity with the simple first touch strategy as OpenMP standard preaches it. Table 2 reports average times for each of the four loop schedulers with two memory distribution strategies. The first strategies, called *numactl* in the table, distributes memory pages in a round robin fashion among the NUMA nodes using `numactl --interleave`. The second strategy leads to a bloc distribution using the API `omp_locality_domain_allocate_bloc1d` presented in section 3.2.

When the array is distributed with the bloc distribution, contigous elements in the array are mostly on the same NUMA node. Even with this distribution, the static scheduler does not reach the best time, which illustrates the irregularity of the application. The dynamic and guided schedulers are able to improve performance but without control of the locality while workload is balanced. The *adaptive* scheduler obtains the best times for the two memory distribution strategies. This is due to a good compromise between a good balance of iterations to control affinity and a dynamic balance of the workload.

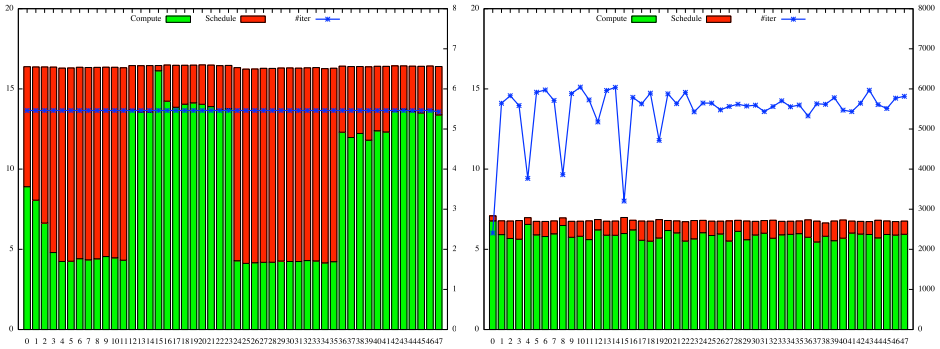
The plots in figures 6, 7 and 8 correspond to execution where the memory is bound using bloc distribution strategy among the NUMA nodes.

Figures 6a and 6b report a per-core execution using the libGOMP *dynamic* and *guided* schedulers with a bloc data distribution over the NUMA nodes. The "compute" (green) part of the graph represents the time spent, in CPU cycles, during the execution of the loop body. The "schedule" part represents the time spent to perform the required computations, apply the scheduling decisions and wait until the loop completion.



(a) Dynamic scheduler, chunk size=3000 (b) Guided scheduler

Fig. 6: Times (ms) per core for the same PMA iteration with dynamic and guided schedulers. The histograms have the same scale.



(a) Static scheduler (b) Adaptive scheduler

Fig. 7: Times (ms) per core for the same PAM iteration with static and adaptive schedulers. The histograms have the same scale.

The histogram represents the number of iterations performed by each core. We can conclude from the top two plots that:

1. the libGOMP *dynamic* scheduler with a chunk size of 1 spends a lot of time in the runtime system. This is mainly due to contention generated by concurrent accesses to internal data structures,
2. the number of iterations and the time spent executing iterations vary from one core to another. The computation of a correct chunk size helps decreasing the schedule time. The performance is thus increased as the average time per iteration decreases from $57.2ms$ to $14.2ms$.

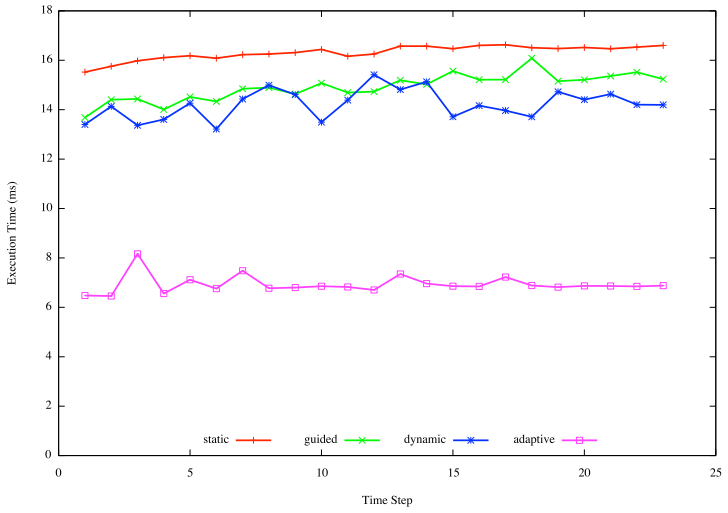


Fig. 8: Comparison of loop schedulers with respect to the time step of PMA simulation.

Figures 7a and 7b report results with the *static* and our *adaptive* loop schedulers with a bloc data distribution. The blue line in the bottom left plot validates the behavior of the static scheduler, as every thread executes the same number of iterations. Nevertheless, the CPU times are highly variable: the distribution of iterations fails at balancing the workload. On the other hand, our *adaptive* scheduler is able to keep the workload well balanced at the expense of an irregular distribution of iterations. The average execution time is $6.95ms$ for the *adaptive* scheduler and $16.3ms$ for the *static* scheduler.

Figure 8 reports the behavior of the 4 loop schedulers on PMA simulation with respect to the time step. Even if the iterations are well balanced among the cores, the *static* scheduler is unable to balance the workload. Both the *dynamic* and the *guided* schedulers reach the same level of performances and are able to better balance the workload, even if memory affinity is not ensured. Our adaptive scheduler is a good compromise between the static and the dynamic schedulers, and reaches a speed-up of 2.35 over the *static* scheduler.

6 Conclusion and Future Work

This paper introduced *adaptive*, a new OpenMP loop scheduler implementing a runtime-level approach to deal with irregular memory-bound applications on NUMA architectures. Instead of distributing statically-fixed portions of work over the threads of a parallel region, this scheduler adapts the granularity of work on demand by making idle threads steal a subset of the victim's remaining iterations, thus introducing the notion of dynamic per-thread granularity. Our scheduler is also capable of adapting its work-stealing algorithm to fit different memory bindings on NUMA architectures and outperforms OpenMP-based approaches to deal with memory locality, like the joint use of the *static* loop scheduling and the *first-touch* allocation policy, on several benchmarks and applications.

This work could be extended to task parallelism, providing the OpenMP application programmer with ways of annotating *splitter* functions called to generate parallelism on demand by splitting a running task into smaller ones. We also consider extending our approach using the concept of *places* recently added to the OpenMP standard that could help the programmer transmitting valuable and portable information on the way memory should be allocated on hierarchical architectures.

Acknowledgement

This work has been partially supported by the ANR 09-COSI-011-05 Project Repdyn and the FP7-PEOPLE-2011-IRSES Project HPC-GA (295217).

References

1. E. Ayguadé, B. Blainey, A. Duran, J. Labarta, F. Martínez, X. Martorell, and R. Silvera. Is the schedule clause really necessary in openmp? In *Proceedings of the OpenMP applications and tools 2003 international conference on OpenMP shared memory parallel programming*, WOMPAT'03, pages 147–159, Berlin, Heidelberg, 2003. Springer-Verlag.

2. F. Broquedis, O. Aumage, B. Goglin, S. Thibault, P.-A. Wacrenier, and R. Namyst. Structuring the execution of OpenMP applications for multicore architectures. In *Proceedings of 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS'10)*, Atlanta, GA, April 2010. IEEE Computer Society Press.
3. F. Broquedis, T. Gautier, and V. Danjean. Libkomp, an efficient openmp runtime system for both fork-join and data flow paradigms. In *Proceedings of the 8th international conference on OpenMP in a Heterogeneous World*, IWOMP'12. Springer-Verlag, 2012.
4. F. Broquedis, N. Furmento, B. Goglin, R. Namyst, and P.-A. Wacrenier. Dynamic Task and Data Placement over NUMA Architectures: an OpenMP Runtime Perspective. In *International Workshop on OpenMP (IWOMP)*, Dresden, Allemagne, 2009.
5. J. M. Bull. Measuring synchronisation and scheduling overheads in openmp. In *In Proceedings of First European Workshop on OpenMP*, pages 99–105, 1999.
6. S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 44–54, Washington, DC, USA, 2009. IEEE Computer Society.
7. M. Durand, B. Raffin, and F. Faure. A Packed Memory Array to Keep Moving Particles Sorted. In *9th Workshop on Virtual Reality Interaction and Physical Simulation*, 2012.
8. M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5):212–223, May 1998.
9. R. C. Hoetzlein. Fluids v2.0, open source, fluid simulator, 2008.
10. L. Huang, H. Jin, L. Yi, and B. Chapman. Enabling locality-aware computations in openmp. *Sci. Program.*, 18(3-4):169–181, August 2010.
11. M. Ihmsen, N. Akinci, M. Becker, and M. Teschner. A parallel sph implementation on multicore cpus. *Computer Graphics Forum*, 30(1):99–112, 2011.
12. A. Mahéo, S. Koliaï, P. Carribault, M. Pérache, and W. Jalby. Adaptive openmp for large numa nodes. In *Proceedings of the 8th international conference on OpenMP in a Heterogeneous World*, IWOMP'12, pages 254–257, Berlin, Heidelberg, 2012. Springer-Verlag.
13. A. Marowka, Z. Liu, and B. Chapman. Openmp-oriented applications for distributed shared memory architectures: Research articles. *Concurr. Comput. : Pract. Exper.*, 2004.
14. J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.
15. S. L. Olivier, B. R. de Supinski, M. Schulz, and J. F. Prins. Characterizing and mitigating work time inflation in task parallel programs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 65:1–65:12, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.
16. S. L. Olivier, A. K. Porterfield, K. B. Wheeler, M. Spiegel, and J. F. Prins. Openmp task scheduling strategies for multicore numa systems. *Int. J. High Perform. Comput. Appl.*, 26(2):110–124, May 2012.
17. OpenMP Architecture Review Board. http://www.openmp.org, 1997-2008.
18. S. Subramaniam and D. L. Eager. Affinity scheduling of unbalanced workloads. In *Proceedings of the 1994 ACM/IEEE conference on Supercomputing*, Supercomputing '94, pages 214–226, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.
19. M. Tchiboukdjian, V. Danjean, T. Gautier, F. Le Mentec, and B. Raffin. A work stealing scheduler for parallel loops on shared cache multicores. In *Proceedings of the 2010 conference on Parallel processing*, Euro-Par 2010, pages 99–107. Springer-Verlag, 2011.
20. D. Traoré, J.-L. Roch, N. Maillard, T. Gautier, and J. Bernard. Deque-free work-optimal parallel stl algorithms. In *Proceedings of the 14th international Euro-Par conference on Parallel Processing*, Euro-Par '08, pages 887–897, Berlin, Heidelberg, 2008.

21. Y. Yan, C. Jin, and X. Zhang. Adaptively scheduling parallel loops in distributed shared-memory systems. *IEEE Trans. on Parallel and Distributed Systems*, 8(1), January 1997.